Exemple de vectorisation avec la factorisation matricielle de Cholesky en Python et Numpy

Peter Philippe

03/06/2025

Introduction

Le principe de la vectorisation consiste à remplacer, lorsque cela est possible, les boucles par des opérations sur des vecteurs, évitant ainsi d'itérer avec des instructions `for` sur les lignes et les colonnes de la matrice. Le code s'en retrouve généralement plus rapide et gagne en lisibilité.

Pour en savoir bien d'avantage sur l'historique de cette méthode, je ne peux que vous conseiller de lire l'excellent article de Claude Brezinski qui s'intitule sobrement « **La méthode de Cholesky** » (ce fichier PDF est librement téléchargeable en cliquant ici).

1 Algorithme de la factorisation de Cholesky

Cet algorithme, qui a plus d'un siècle, porte le nom de son inventeur André-Louis Cholesky (1875-1918), il est de type direct, s'applique à des matrices carrées $\mathbf{A}$, symétriques $a_{i,j} = a_{j,i}$, définies positives (c'est-à-dire dont toutes les valeurs propres sont positives) et permet une factorisation telle que $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$. Cette matrice $\mathbf{L}$ est unique et triangulaire inférieure, tous les éléments de sa diagonale sont strictements positifs. Comme pour toute matrice triangulaire à diagonale non nulle $\ell_{i,i} \neq 0$, elle est de rang plein, donc inversible et son déterminant non nul vaut $|\mathbf{L}| = \prod_{i}^{n} \ell_{i,i}^2$. Contrairement à la méthode d'élimination de Gauss, aucun changement de ligne n'est effectué, sans quoi la structure symétrique de la matrice en serait irrémédiablement détruite.

Une variante permettant d'obtenir $\mathbf{L}$ supérieure existe, ainsi que d'autres, notamment l'algorithme $\mathbf{A} = \mathbf{LDL}^{\mathrm{T}}$ dont la particularité est de ne plus utiliser la racine carrée, une version adaptée pour les matrices par blocs ou encore lorsque les coefficients de $\mathbf{A}$ sont dans $\mathbb{C}$.
Cet algoritme est aussi applicable lorsque la matrice est creuse (ceci est particulièrement utile pour le préconditionnement de la méthode du gradient conjugué), il se nomme « *factorisation incomplète de Cholesky* », il évite ainsi le phénomène de *fill-in* dans la partie triangulaire utile de $\mathbf{L}$, c'est-à-dire d'éviter l'ajout des valeurs non nulles là où il ne doit pas y en avoir. Le produit $\mathbf{LL}^{\mathrm{T}}$ permet alors de respecter la structure creuse originelle [1] de $\mathbf{A}$.

Étant donné la présence d'une racine carrée, cet algorithme échouera si l'un des éléments de la diagonale n'est pas strictement positif, strictement ? oui, car la valeur de cette racine carrée sert ensuite de dénominateur :

$$l_{i,j} = \begin{cases} l_{1,1} & \sqrt{a_{1,1}} \\ 0 & i < j \\ \sqrt{a_{j,j} - \sum_{k=1}^{j-1} (l_{j,k})^2} & i = j \\ \frac{1}{l_{j,j}} \left(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} l_{j,k} \right) & i > j \quad i = (j+1),(j+2),\ \ldots \end{cases} \tag{1}$$

ceci implique que la valeur sous la racine carrée doit toujours être strictement positive.

Pour une matricette $\mathbf{A}$ d'ordre 3, la factorisation de Cholesky aboutira à la matrice $\mathbf{L}$:

$$\mathbf{L} = \begin{pmatrix} l_{1,1} & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} \end{pmatrix} = \begin{pmatrix} \sqrt{a_{1,1}} & 0 & 0 \\ \frac{a_{2,1}}{l_{1,1}} & \sqrt{a_{2,2} - l_{2,1}^2} & 0 \\ \frac{a_{3,1}}{l_{1,1}} & \frac{a_{3,2} - l_{2,1} l_{3,1}}{l_{2,2}} & \sqrt{a_{3,3} - l_{3,1}^2 - l_{3,2}^2} \end{pmatrix}$$

Le produit $\mathbf{LL}^{\mathrm{T}}$ permettant de retrouver $\mathbf{A}$:

$$\mathbf{A} = \mathbf{LL}^{\mathrm{T}} = \begin{pmatrix} l_{1,1} & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} \end{pmatrix} \begin{pmatrix} l_{1,1} & l_{2,1} & l_{3,1} \\ 0 & l_{2,2} & l_{3,2} \\ 0 & 0 & l_{3,3} \end{pmatrix} = \begin{pmatrix} l_{1,1}^2 & l_{1,1} l_{2,1} & l_{1,1} l_{3,1} \\ l_{1,1} l_{2,1} & l_{2,1}^2 + l_{2,2}^2 & l_{2,1} l_{3,1} + l_{2,2} l_{3,2} \\ l_{1,1} l_{3,1} & l_{2,1} l_{3,1} + l_{2,2} l_{3,2} & l_{3,1}^2 + l_{3,2}^2 + l_{3,3}^2 \end{pmatrix}$$

1. ou *Sparsity pattern*

2 Implémentation en Python

Ce petit code comprend deux versions, la première est bien classique puisqu'il s'agit de celle par boucle et la seconde est vectorisée via l'emploi de la fonction `np.sum()` de Numpy. L'algorithme de Cholesky se prête très bien pour cet exemple, mais en pratique, il faudra évidemment choisir la fonction `np.linalg.cholesky()` qui se montrera plus véloce ou mieux, son homologue de la librairie Cupy :

```
import numpy as np
import time as t

def chrono(f: callable) -> float:
    def appel(*args, **kargs) -> None:
        debut = t.perf_counter()
        result = f(*args, **kargs)
        fin = t.perf_counter() - debut
        print(f"\t-> Temps de calcul de la fonction {f.__name__}() : {fin:.2f} s")
        return result
    return appel


def test_matrice(a: np.ndarray) -> None:
    if a.ndim != 2:
        raise np.linalg.LinAlgError("A n'est pas en 2D !")

    if a.size == 0:
        raise np.linalg.LinAlgError("A est vide !")

    n, m = a.shape
    if n != m:
        raise np.linalg.LinAlgError("A est rectangulaire !")

    valp = np.linalg.eigvals(a)
    if np.allclose(a, a.T) == False and np.any(valp) <= 0.0:
        raise np.linalg.LinAlgError("A n'est SDP")


@chrono
def cholesky(a: np.array) -> np.ndarray:
    test_matrice(a)
    L = np.copy(a)
    n = L.shape[0]

    for i in range(n):
        tmp = L[i,i]
        L[i,i] = np.sqrt(tmp)

        # éléments de la diagonale
        for j in range(i+1,n):
            L[j,i] /= L[i,i]
```

```
        # éléments hors diagonale
        for k in range(i+1,n):
            L[i,k] = 0
            for j in range(k,n):
                L[j,k] -= L[j,i] * L[k,i]

    return L


@chrono
def cholesky_vectorisee(a: np.ndarray) -> np.ndarray:
    test_matrice(a)
    L = np.copy(a)
    n = L.shape[0]

    for i in range(n):
        # éléments de la diagonale
        L[i,i] = np.sqrt(a[i,i] - np.sum(L[i,:i]**2))

        # éléments hors diagonale
        for j in range(i+1, n):
            L[j,i] = (a[j,i] - np.sum(L[j,:i] * L[i,:i])) / L[i,i]

    return L


def creer_matrice_spd(n: int) -> np.ndarray:
    rng = np.random.default_rng()
    m: np.ndarray = rng.random(size=(n,n), dtype=np.float32)

    a = m @ m.T
    d = np.diag(np.diag(a))
    a += d * 2

    return a


if __name__ == "__main__":
    for n in [100, 200, 400, 800, 1600, 3200]:
        matrice = creer_matrice_spd(n, False)
        print(f"Pour une matrice d'ordre {n} :")
        cholesky(matrice)
        cholesky_vectorisee(matrice)
```

Plus l'ordre augmente, plus les écarts des temps d'exécutions mesurés sont impressionnants (le CPU utilisé est un Intel Core i5 13600KF) :

```
% Pour une matrice d'ordre 100 :

        -> Temps de calcul de la fonction cholesky() : 0.04 s
```

```
       -> Temps de calcul de la fonction cholesky_vectorisee() : 0.02 s

Pour une matrice d'ordre 200 :

       -> Temps de calcul de la fonction cholesky() : 0.33 s

       -> Temps de calcul de la fonction cholesky_vectorisee() : 0.06 s

Pour une matrice d'ordre 400 :

       -> Temps de calcul de la fonction cholesky() : 2.52 s

       -> Temps de calcul de la fonction cholesky_vectorisee() : 0.25 s

Pour une matrice d'ordre 800 :

       -> Temps de calcul de la fonction cholesky() : 19.90 s

       -> Temps de calcul de la fonction cholesky_vectorisee() : 1.03 s

Pour une matrice d'ordre 1600 :

       -> Temps de calcul de la fonction cholesky() : 158.69 s

       -> Temps de calcul de la fonction cholesky_vectorisee() : 4.17 s

Pour une matrice d'ordre 3200 :

       -> Temps de calcul de la fonction cholesky() : 1277.60 s

       -> Temps de calcul de la fonction cholesky_vectorisee() : 20.41 s
```

Le petit graphique permettra de mieux se rendre compte de la différence de vitesse, il a été généré avec le code ci-dessous incluant la librairie Matplotlib :

```
import matplotlib.pyplot as mpl

ordres = [100, 200, 400, 800, 1600, 3200]
chronos_cholesky = [0.04, 0.33, 2.52, 19.90, 158.69, 1277.60]
chronos_cholesky_vectorisee = [0.02, 0.06, 0.25, 1.03, 4.17, 20.41]

mpl.figure(figsize=(12, 7))
mpl.plot(ordres,
        chronos_cholesky,
        marker="o",
        ls="-",
        color="#ff3333",
        markersize=7,
        lw=2,
        label="Cholesky")
```

```
mpl.plot(ordres,
        chronos_cholesky_vectorisee,
        marker="x",
        ls="--",
        color="#33ff00",
        markersize=7,
        lw=2,
        label="Cholesky vectorisée")

police = {"family": "serif", "color": "k", "weight": "bold", "size": 12 }
mpl.xlabel("Ordre de la matrice (100, 200, 400, 800, 1600, 3200)", fontdict=police)
mpl.ylabel("Temps d'éxecution en secondes", fontdict=police)
mpl.title("Comparaison entre la version boucle et la version vectorisée", fontdict=police, pad=20)
mpl.legend(fontsize=10, frameon=True, shadow=True, fancybox=True)
mpl.grid(True, which="both", ls=":", lw=0.5, color="gray", alpha=0.7)
mpl.tick_params(axis="both", which="major", labelsize=12, grid_linewidth=1, grid_linestyle="-")
mpl.tight_layout()
mpl.minorticks_on()
mpl.gca().set_aspect("equal", adjustable="box")
mpl.show()
```